

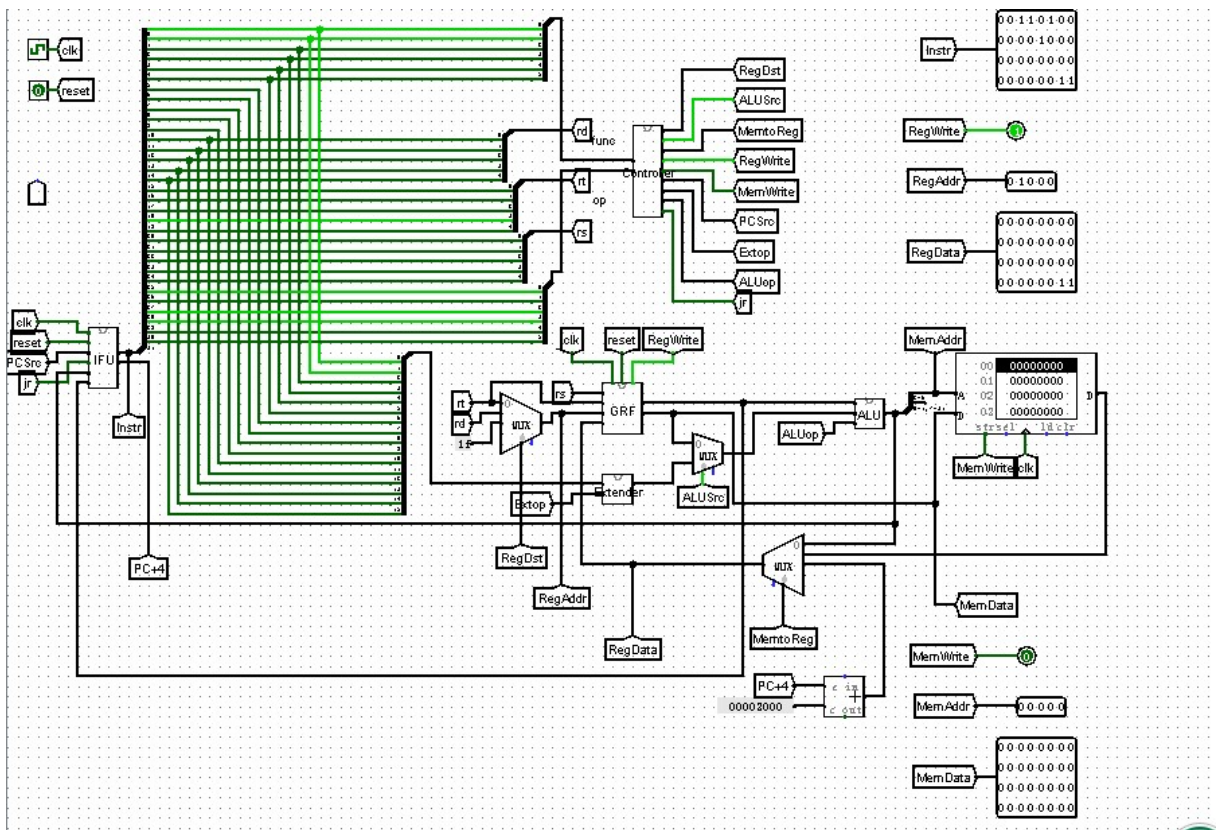
利用 Verilog 开发 MIPS 单周期处理器

一、整体结构：

控制器（Controller）、IFU（取指令单元）、GRF（通用寄存器组，也称为寄存器文件、寄存器堆）、ALU（算术逻辑单元）、DM（数据存储器）、EXT（位扩展器）。

处理器为 32 位处理器

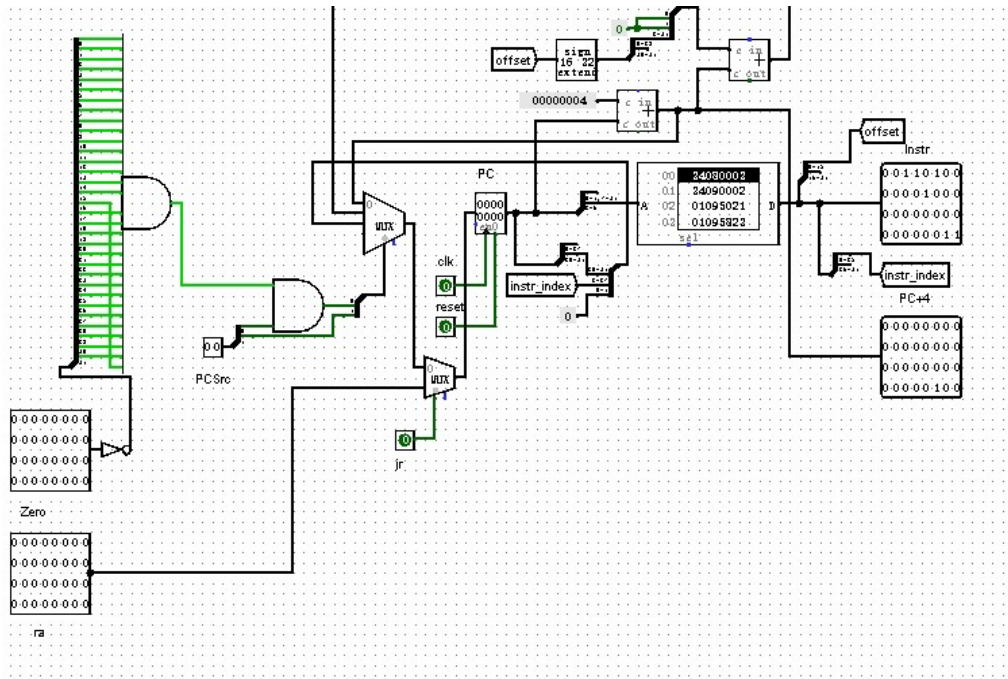
处理器应支持的指令集为：{addu, subu, ori, lw, sw, beq, lui, jal, jr, nop}。



二、数据通路设计：

（一）模块规格撰写

1. IFU（取指令单元）：内部包括 PC、IM、NPC



1) PC（程序计数器）

器件：32bit 寄存器

模块接口

信号名	方向	描述
NPC[31:0]	I	下一条指令的地址
clk	I	时钟信号
reset	I	复位信号 1: 有效 0: 无效
PC[31:0]	O	当前指令地址

功能定义

序号	功能名称	功能描述
1	复位	当复位信号有效时，PC 被设置为起始地址 0x00003000

2) IM (指令存储器)

IM 容量为 4KB (32bit×1024 字)

因为 ROM 中储存了 1024 个地址，且 IM 实际地址宽度仅为 10 位，，从而将地址的低 10 位 (2~11 位) 连接到 ROM 选择地址端口。

模块接口

信号名	方向	描述
addr[31:0]	I	当前指令的地址
op[5:0]	O	6 位 op 信号
func[5:0]	O	6 位 func 信号
rs	O	rs 寄存器编号
rt	O	rt 寄存器编号
rd	O	rd 寄存器编号
imm[15:0]	O	16 位 offset
imm26[25:0]	O	26 位 instr_index

3) NPC (计算下一条指令)

器件：加法器、多路选择器

在处理器支持的指令集中，下一条指令的选择分为两种情况：

- 1) addu, subu, ori, lw, sw, lui, nop: $PC = PC + 4$
- 2) beq: $PC = (PC + 4) + \text{sign_extend}(\text{imm16}) < 2$
- 3) jal: $PC = \{PC[31:28], \text{instr_index}, 2'b00\}$

4) jr: $PC = GPR[rs]$

模块接口

信号名	方向	描述
PC[31:0]	I	当前指令地址
imm[15:0]	I	16 位 offset
imm26[25:0]	I	26 位 instr_index
jr_ra[31:0]	I	jr 指令时, rs 寄存器中的值
PCSrc[1:0]	I	PC 选择信号 01: 当前指令为 beq 10: 当前指令为 jal 00: 其他指令
jr	I	jr 指令信号
zero	I	ALU 计算结果为 0 标志 0: 计算结果为 0 非 0: 计算结果非 0
NPC[31:0]	O	下一条指令的地址
PCplus4	O	PC+4 的值

功能定义

序号	功能名称	功能描述
1	复位	当复位信号有效时, PC 被设置为起始地址 0x00000000
3	计算下一条指令地址	PCSrc=00 且 jr=0 时: $PC = PC + 4$

		PCSrc=01 且 jr=0 时: $PC=(PC+4) + \text{sign_extend}(\text{imm16}) \ll 2$ PCSrc=10 且 jr=0 时: $PC=\{PC[31:28], \text{instr_index}, 2'b00\}$ jr=1 时: $PC=GPR[rs]$
--	--	---

2.GRF（通用寄存器组）：内部包括 32 个寄存器

具有写使能的寄存器实现，寄存器总数为 32 个

0 号寄存器的值始终保持为 0。其他寄存器初始值均为 0

模块接口

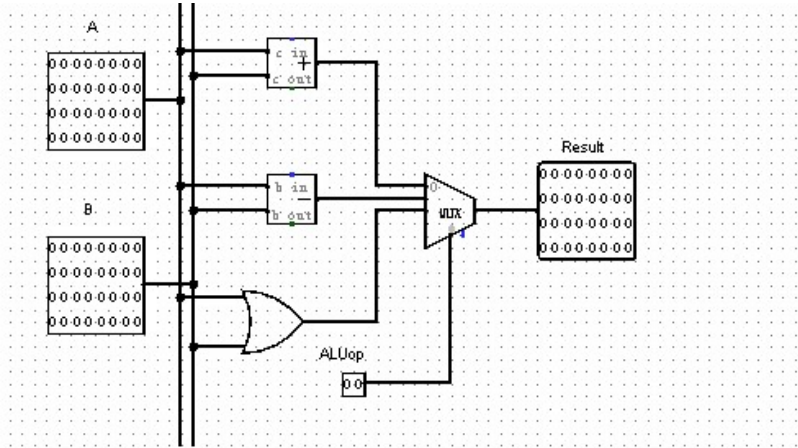
信号名	方向	描述
WPC[31:0]	I	相应指令存储地址
clk	I	时钟信号
reset	I	复位信号 1：有效 0：无效
RegWrite	I	读写控制信号 1：写操作 0：读操作
Read_register1[4:0]	I	读寄存器 1 的地址
Read_register2[4:0]	I	读寄存器 2 的地址
Write_register[4:0]	I	写寄存器的地址
Write_data[31:0]	I	向写寄存器中写入的值
Read_data1[31:0]	O	32 位输出 1

Read_data2[31:0]	O	32 位输出 2
------------------	---	----------

功能定义

序号	功能名称	功能描述
1	复位	当复位信号有效时，所有寄存器的值被设置为 0x00000000
2	写寄存器	根据输入的写寄存器地址，把输入的数据写入写寄存器中
3	读寄存器	根据输入的读寄存器地址，将数据读出

3. ALU（算术逻辑单元）



提供 32 位加、减、或运算

可以不支持溢出

模块接口

信号名	方向	描述
A[31:0]	I	ALU32 位输入数据 A
B[31:0]	I	ALU32 位输入数据 B
ALUop[1:0]	I	ALU 功能选择信号 00:加法

		01:减法 10:或运算
Result[31:0]	O	32 位数据输出

功能定义

序号	功能名称	功能描述
1	或	A B
2	减	A-B
3	加	A+B

4.DM（数据存储器）

DM 容量为 4KB（32bit×1024 字）

起始地址：0x00003000

模块接口

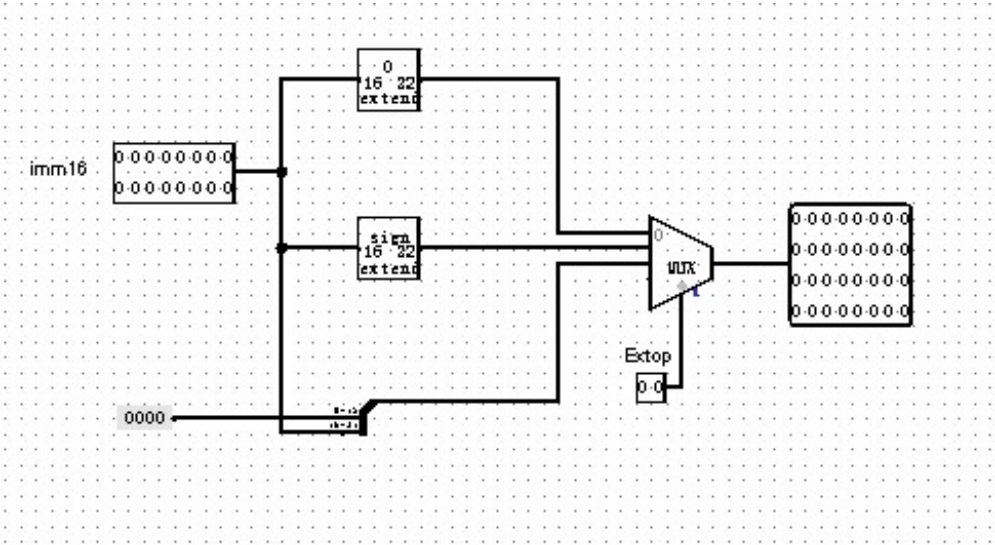
信号名	方向	描述
pc[31:0]	I	对应的 pc
addr[31:0]	I	对应指令的地址
clk	I	时钟信号
reset	I	复位信号 1：有效 0：无效
MemData[31:0]	I	写入数据的输入

MemWrite	I	读写控制信号 1: 写操作
out	O	读取数据的输出

功能定义

序号	功能名称	功能描述
1	复位	当复位信号有效时,所有数据被设置为 0x00000000
2	写操作	根据输入的寄存器地址,把输入的数据写入
3	读操作	根据输入的寄存器地址,将其中的数据读出

5. EXT（位扩展器）：



使用 logisim 内置的 Bit Extender

模块接口

信号名	方向	描述
imm[15:0]	I	16 位 imm 数据输入

Extop[1:0]	I	位扩展选择信号 00: 高位补 0 01: 高位符号扩展 10: 低位补 0
after_ext[31:0]	O	位扩展后的 32 位输出

功能定义

序号	功能名称	功能描述
1	高位补 0	高 16 位补 0
2	低位补 0	低 16 位补 0
3	符号扩展	若符号位为 0，则高位补 0 若符号位为 1，则高位补 1

6.控制器（Controller）

模块接口

信号名	方向	描述
func[5:0]	I	6 位 function
op[5:0]	I	6 位 op
RegDst[1:0]	O	写寄存器地址控制
ALUsrc	O	ALU 的 B 操作数的选择控制
MemtoReg[1:0]	O	写寄存器的数据来源选择控制
RegWrite	O	GRF 读写控制信号

MemWrite	O	DM 写控制信号，写入 GRF 的数据选择
PCsrc[1:0]	O	PC 选择信号
Extop[1:0]	O	控制扩展方式
ALUSrc[1:0]	O	控制 ALU 进行相应运算
jr	O	jr 信号

6. mux（多路选择器）

1)mux2_32

信号名	方向	描述
Control	I	控制选择信号
din0[31:0]	I	Control=0 时的选择
din1[31:0]	I	Control=1 时的选择
out[31:0]	O	选择的输出

2)mux3_5

信号名	方向	描述
Control[1:0]	I	控制选择信号
din0[4:0]	I	Control=00 时的选择
din1[4:0]	I	Control=01 时的选择
din2[4:0]	I	Control=10 时的选择
out[4:0]	O	选择的输出

3)mux3_5

信号名	方向	描述
-----	----	----

Control[1:0]	I	控制选择信号
din0[31:0]	I	Control=00 时的选择
din1[31:0]	I	Control=01 时的选择
din2[31:0]	I	Control=10 时的选择
out[31:0]	O	选择的输出

（二）思考题

1. 根据你的理解，在下面给出的 DM 的输入示例中，地址信号 addr 位数为什么是[11:2]而不是[9:0]？这个 addr 信号又是从哪里来的？

文件	模块接口定义
dm.v	<pre>dm(clk,reset,MemWrite,addr,din,dout); input clk; //clock input reset; //reset input MemWrite; //memory write enable input [11:2] addr; //memory's address for write input [31:0] din; //write data output [31:0] dout; //read data</pre>

答：在 DM 中，内存是按字划分的。一个字是四字节，因此在进行 DM 中进行内存的存取时，应当将输入地址除以 4（或者右移两位），作为实际地址，即取 2~11 位。

addr 信号来自 ALU 的运算结果。

2. 在相应的部件中，reset 的优先级比其他控制信号（不包括 clk 信号）都要高，且相应的设计都是同步复位。清零信号 reset 是针对哪些部件进行清零复位操作？这些部件为什么需要清零？

答：reset 针对 PC、GRF、DM，其中 PC 需要回到初始指令地址 0x00003000，GRF 需要清空全部寄存器的值，DM 也清空。

(三) IM 设计

```
module IM(  
  
    input[31:0] addr,  
  
    output [5:0] op,  
  
    output [5:0] func,  
  
    output [4:0] rs,  
  
    output [4:0] rt,  
  
    output [4:0] rd,  
  
    output [15:0] imm,  
  
    output [25:0] imm26  
  
);  
  
wire[9:0] address;  
  
reg[31:0] im[1023:0];    //ROM  
  
assign address[9:0]=addr[11:2];  
  
initial begin  
  
    $readmemh("code.txt",im);    //指令放入 im  
  
end  
  
assign op[5:0]=im[address][31:26];  
  
assign func[5:0]=im[address][5:0];
```

```

assign rs[4:0]=im[address][25:21];

assign rt[4:0]=im[address][20:16];

assign rd[4:0]=im[address][15:11];

assign imm[15:0]=im[address][15:0];

assign imm26[25:0]=im[address][25:0];

```

```
endmodule
```

（四）控制器设计

（一）设计方式

1. 数据通路设计

指令	Adder		PC	IM Add res s	Registers				ALU		DM		Ext	Nadd	
	A	B			Reg 1	Reg 2	Wre g	Wdat a	A	B	Addre ss	Wdata			
addu/subu	PC	4	Adder	PC	Rs	Rt	Rd	ALU	Rdata1	Rdata2					
Ori	PC	4	Adder	PC	Rs		Rt	ALU	Rdata1	Ext			imm16		
Lw	PC	4	Adder	PC	Rs		Rt	DM	Rdata1	Ext	ALU		imm16		
Sw	PC	4	Adder	PC	Rs	Rt			Rdata1	Ext	ALU	Rdata2	imm16		
Beq	PC	4	Adder Nadd	PC	Rs	Rt			Rdata1	Rdata2			imm16	Adder	Shift
Lui	PC	4	Adder	PC	\$0		Rt	ALU	Rdata1	Ext			imm16		
Nop															
合并	PC	4	Adder Nadd	PC	Rs	Rt	Rt Rd	ALU DM	Rdata1	Rdata2 Ext	ALU	Rdata2	imm16	Adder	Shift

2. 主控单元真值表

	addu	subu	or i	lw	sw	beq	lui
op	000000	000000	00110	10001	10101	00010	00111
			1	1	1	0	1

func	100001	100011					
RegDst	01	01	00	00	00	00	00
ALUSrc	0	0	1	1	1	0	1
MemtoReg	00	00	00	01	00	00	00
RegWrite	1	1	1	1	0	0	1
MemWrite	0	0	0	0	1	0	0
PCSrc	00	00	00	00	00	01	00
Extop	xx	xx	00	01	01	01	10
ALUop	00	01	10	1	00	01	10
jr	0	0	0	0	0	0	0
	jal	jr					
op	000011	000000					
func		001000					
RegDst	10	00					
ALUSrc	x	0					
MemtoReg	10	00					
RegWrite	1	0					
MemWrite	0	0					
PCSrc	10	00					
Extop	00	00					
ALUop	00	00					

jr	0	1					
----	---	---	--	--	--	--	--

（二）思考题

1. 列举出用 Verilog 语言设计控制器的几种编码方式（至少三种），并给出代码示例。

答：

1) 利用 if-else（或 case）完成操作码和控制信号的值之间的对应；

```
always@(*)begin

    case(op)

        6'b000000:

            begin

                if(func==6'b001000)begin        //jr

                    RegDst=2'b00;

                    ALUSrc=0;

                    MemtoReg=2'b00;

                    RegWrite=0;

                    MemWrite=0;

                    PCSrc=2'b00;

                    Extop=2'b00;

                    ALUOp=2'b00;

                    jr=1;

                    beq=0;
```

```

end

else if(func==6'b100001)begin    //addu

    RegDst=2'b01;

    ALUSrc=0;

    MemtoReg=2'b00;

    RegWrite=1;

    MemWrite=0;

    PCSrc=2'b00;

    Extop=2'b00;

    ALUOp=2'b00;

    jr=0;

    beq=0;

end

else if(func==6'b100011)begin    //subu

    RegDst=2'b01;

    ALUSrc=0;

    MemtoReg=2'b00;

    RegWrite=1;

    MemWrite=0;

    PCSrc=2'b00;

    Extop=2'b00;

    ALUOp=2'b01;

```



```

        jr=0;

        beq=0;

    end

    else begin

        RegDst=2'b00;

        ALUSrc=0;

        MemtoReg=2'b00;

        RegWrite=0;

        MemWrite=0;

        PCSrc=2'b00;

        Extop=2'b00;

        ALUOp=2'b00;

        jr=0;

        beq=0;

    end

end

```

2) 利用 assign 语句完成操作码和控制信号的值之间的对应;

```

assign RegDst[0] = ( (op == 6'b000000 && func == 6'b100001) || (op == 6'b000000
&& func == 6'b100011) )? 1 : 0 ;

```

3) 利用宏定义

```

`define state1 4'b0001

```

```
`define state2 4'b0010
```

```
`define state3 4'b0100
```

```
`define state4 4'b1000
```

`define 标识符(宏名) 字符串(宏内容)

如: **`define signal string**

它的作用是指定用标识符 **signal** 来代替 **string** 这个字符串, 在编译预处理时, 把程序中在该命令以后所有的 **signal** 都替换成 **string**。这种方法使用户能以一个简单的名字代替一个长的字符串, 也可以用有一个有含义的名字来代替没有含义的数字和符号, 因此把这个标识符(名字)称为“宏名”, 在编译预处理时将宏名替换成字符串的过程称为“宏展开”。**`define** 是宏定义命令。

2. 根据你所列举的编码方式, 说明他们的优缺点。

编码方式	优点	缺点
if-else/case	代码写起来比较容易、直观, 与 C 语言类似	它们在不注意的情况下容易产生锁存器, 对毛刺敏感, 使其处于不确定状态, 且语句较为冗长
assign	语句较为简单, 看起来比较方便	适用范围比较窄, 只能对 wire 型变量赋值
宏定义	增加可读性、可修改性好、设计的可重用性好	定义时容易出错, 且不容易查错

(五) 在线测试相关信息

(一) 测试代码

```

1  .data
2  .text
3  ori $t0,$0,0
4  ori $t1,$0,1 # $t1 = 1
5  ori $t2,$0,2 # $t2 = 2
6
7  beq $t1,$0, jump1
8  ori $t0,$0,1 # $t0 = 1
9  jump1:
10 beq $t0,$0, jump2
11 ori $t0,$0,0
12 jump2:
13
14 jump3:
15 addu $t3,$t1,$t2 # $t3 = 3
16 subu $t4,$t2,$t1 # $t4 = 1
17 sw $t3,0($sp)
18 sw $t4,-4($sp)
19 lw $t5,0($sp)

```

```

20 lw $t6,-4($sp)
21 lui $t7,15
22 beq $t1,$t2, jump3
23
24 jal jump4
25 jump5:
26 beq $t0,$t2, over1
27 beq $t0,$t3, over2
28 jump4:
29 ori $t0,$0,2
30 beq $t0,$t2, jump5
31
32 over1:
33 ori $t0,$0,3
34 jr $ra
35
36 over2:
37 ori $t0,$0,4

```

Edit		Execute		
Text Segment				
Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00003000	0x34080000	ori \$8,\$0,0x00000000	3: ori \$t0,\$0,0
<input type="checkbox"/>	0x00003004	0x34090001	ori \$9,\$0,0x00000001	4: ori \$t1,\$0,1 # \$t1 = 1
<input type="checkbox"/>	0x00003008	0x340a0002	ori \$10,\$0,0x00000002	5: ori \$t2,\$0,2 # \$t2 = 2
<input type="checkbox"/>	0x0000300c	0x11200001	beq \$9,\$0,0x00000001	7: beq \$t1,\$0, jump1
<input type="checkbox"/>	0x00003010	0x34080001	ori \$8,\$0,0x00000001	8: ori \$t0,\$0,1 # \$t0 = 1
<input type="checkbox"/>	0x00003014	0x11000001	beq \$8,\$0,0x00000001	10: beq \$t0,\$0, jump2
<input type="checkbox"/>	0x00003018	0x34080000	ori \$8,\$0,0x00000000	11: ori \$t0,\$0,0
<input type="checkbox"/>	0x0000301c	0x012a5821	addu \$11,\$9,\$10	15: addu \$t3,\$t1,\$t2 # \$t3 = 3
<input type="checkbox"/>	0x00003020	0x01496023	subu \$12,\$10,\$9	16: subu \$t4,\$t2,\$t1 # \$t4 = 1
<input type="checkbox"/>	0x00003024	0xafab0000	sw \$11,0x00000000(\$29)	17: sw \$t3,0(\$sp)
<input type="checkbox"/>	0x00003028	0xafacffff	sw \$12,0xffffffff(\$29)	18: sw \$t4,-4(\$sp)
<input type="checkbox"/>	0x0000302c	0x8fad0000	lw \$13,0x00000000(\$29)	19: lw \$t5,0(\$sp)
<input type="checkbox"/>	0x00003030	0x8faefffc	lw \$14,0xffffffff(\$29)	20: lw \$t6,-4(\$sp)
<input type="checkbox"/>	0x00003034	0x3c0f000f	lui \$15,0x0000000f	21: lui \$t7,15
<input type="checkbox"/>	0x00003038	0x112afff8	beq \$9,\$10,0xffffffff8	22: beq \$t1,\$t2, jump3
<input type="checkbox"/>	0x0000303c	0x0c000c12	jal 0x00003048	24: jal jump4
<input type="checkbox"/>	0x00003040	0x110a0003	beq \$8,\$10,0x00000003	26: beq \$t0,\$t2, over1
<input type="checkbox"/>	0x00003044	0x110b0004	beq \$8,\$11,0x00000004	27: beq \$t0,\$t3, over2
<input type="checkbox"/>	0x00003048	0x34080002	ori \$8,\$0,0x00000002	29: ori \$t0,\$0,2
<input type="checkbox"/>	0x0000304c	0x110afffc	beq \$8,\$10,0xffffffffc	30: beq \$t0,\$t2, jump5
<input type="checkbox"/>	0x00003050	0x34080003	ori \$8,\$0,0x00000003	33: ori \$t0,\$0,3
<input type="checkbox"/>	0x00003054	0x03e00008	jr \$31	34: jr \$ra
<input type="checkbox"/>	0x00003058	0x34080004	ori \$8,\$0,0x00000004	37: ori \$t0,\$0,4

34080000

34090001

340a0002

11200001

34080001

11000001

34080000

012a5821

01496023

afab0000

afacffff

8fad0000

8faefffc

3c0f000f

112afff8

0c000c12

110a0003

110b0004

34080002

110afffc

34080003

03e00008

34080004

(二) 预期结果

@00003000: \$ 8 <= 00000000
@00003004: \$ 9 <= 00000001
@00003008: \$10 <= 00000002
@00003010: \$ 8 <= 00000001
@00003018: \$ 8 <= 00000000
@0000301c: \$11 <= 00000003
@00003020: \$12 <= 00000001
@00003024: *00000000 <= 00000003
@00003028: *ffffffc <= 00000001
@0000302c: \$13 <= 00000003
@00003030: \$14 <= 00000001
@00003034: \$15 <= 000f0000
@0000303c: \$31 <= 00003040
@00003048: \$ 8 <= 00000002
@00003050: \$ 8 <= 00000003
@00003058: \$ 8 <= 00000004

(三) 思考题

ADD: 符号加

编码	31	26	25	21	20	16	15	11	10	6	5	0
	special 000000	rs		rt		rd		0 00000		add 100000		
	6	5		5		5		5		6		
格式	add rd, rs, rt											
描述	$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$											
操作	<pre>temp ← (GPR[rs]₃₁ GPR[rs]) + (GPR[rt]₃₁ GPR[rt]) if temp₃₂ ≠ temp₃₁ then SignalException(IntegerOverflow) else GPR[rd] ← temp_{31..0} endif</pre>											
示例	add \$s1, \$s2, \$s3											
其他	temp ₃₂ ≠ temp ₃₁ 代表计算结果溢出。 如果不考虑溢出, 则 add 与 addu 等价。											

ADDU: 无符号加

编码	31	26	25	21	20	16	15	11	10	6	5	0
	special 000000	rs	rt	rd	0 00000	addu 100001						
	6	5	5	5	5	6						
格式	addu rd, rs, rt											
描述	$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$											
操作	$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$											
示例	addu \$s1, \$s2, \$s3											
其他												

2. 根据自己的设计说明单周期处理器的优缺点。

答:

优点	缺点
控制部件相比多周期CPU更简单, 且实际设计起来较为简便, 不容易出错	用一个时钟周期执行一条指令, 从而确定时钟周期的时间长度要考虑执行的时间最长的指令, 一次确定 CPU 频率, 不管指令复杂度如何, 单周期 CPU 花费相同时间执行, 这造成时间上浪费

简要说明 jal、jr 和堆栈的关系

答：在跳转到指定地址实现子程序调用的同时，需要将返回地址保存到 ra 寄存器，即通常所说的“函数调用的现场保护”，以便子程序返回时能够继续调用之前的流程。对于跳转/分支指令，MIPS CPU 将自动保存 ra；若子程序需要嵌套调用其他子程序，则必须先存储 ra，通常是压入栈，子程序末尾弹出之前保存的 ra，然后 jr 到 ra。这两条指令分别实现了直接和间接子程序调用。